Chapter 3: Decisions

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

Objectives

In this chapter you will learn about:

- The if Statement
- Logical Operators
- Boolean Expressions and Variables
- Conditional Execution
- String Comparison
- Alternative Execution - if else
- Chained Conditionals – if elif else
- Nested Conditionals
- Catching Exceptions using Try and Except
- Short-Circuit Evaluation of Logical Expressions

Control Structures

A control structure like the if statement allows a program to check conditions and change the behavior of the program. This allows the computer to make decisions based on input.

Boolean Variables

Try each of these exercises at the Python Interactive session.

A Boolean expression is an expression that is either true or false. True and False are so-called "Boolean values" that are predefined in Python. True and False are the only Boolean values, and anything that is not False, is True.

The following examples use a comparison operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the class bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The == operator is one of the comparison operators; the others are:

x != y	x is not equal to y
x > y	x is greater than y
x < y	x is less than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y
x is y	x is the same as y
x is not y	x is not the same as y

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. There is no such thing as =< or =>.

The if Statement

Pseudo-code for an if statement.

```
if expression is True:
    Execute one or more statements at same indentation level
Execute next statement
```

The expression must be one that will evaluate to either True or False.

1. If the expression evaluates to True, the indented statement(s) that follow the colon will be executed in sequence.
2. If the expression evaluates to False, those indented statements will be skipped and the next statement following the if statement will be executed.

Common Mistakes

Mistake 1 A common mistake is to use the assignment statement = instead of the equality statement ==

Incorrect	**Correct**
`if x = 1:`	`if x == 1:`

Mistake 2 A common mistake is to use `and` where `or` is needed or vice-versa. Consider the following if statements:

```
if x > 1 and x < 100:
if x > 1 or x < 100:
```

The first statement is the correct one. If x is any value between 1 and 100, then the statement will be true. The idea is that x must be both greater than 1 and less than 100. On the other hand, the second statement is not what we want because for it to be true, either x has to be greater than 1 or x has to be less than 100. But every number satisfies this. The lesson here is if your program is not working correctly, check your and's and or's.

Mistake 3 Another very common mistake is to write something like below:

```
if grade >= 80 and < 90:
```

This will lead to a syntax error. We must be explicit. The correct statement is:

```
if grade >= 80 and grade < 90:
```

Logical Operators

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is like their meaning in English.

and	And	a and b	true if both a and b are true
or	Inclusive OR	a or b	true if either a or b are true
not	Negation	not a	switch a from true to false or from false to true

For example,

```
x > 0 and x < 10
```

is true only if x is greater than 0 and less than 10.

n % 2 == 0 or n % 3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

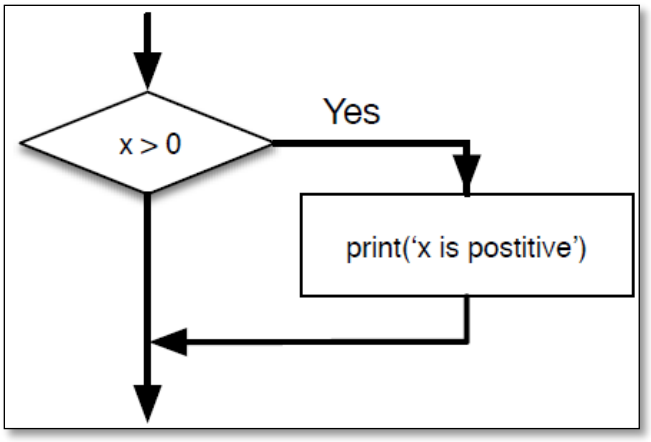
Finally, the not operator negates a Boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.

Conditional Execution

To write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the `if` statement:

```
x = 0
if x > 0:
   print('x is positive')
```

The Boolean expression after the `if` statement is called the condition. We end the `if` statement with a colon character (:) and the line(s) after the if statement are indented.



If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

`if` statements have the same structure as function definitions or for loops. The statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called compound statements because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0:
    pass    # TODO: Need to handle negative values!
```

Tutorial 3.1 – Guessing Game

Let's try a guess-a-number program. The computer picks a random number, the player tries to guess, and the program tells them if they are correct. To see if the player's guess is correct, we need an if statement.

The syntax of the `if` statement is a lot like the for statement in that there is a colon at the end of the if condition and the following line or lines are indented. The lines that are indented will be executed only if the condition is true. Once the indentation is done with, the `if` block is concluded.

Create and test the following program called **guessing_game.py**

```
"""
    Name: guessing_game.py
    Author:
    Created:
    Purpose: Demonstrate if else and random numbers
"""

# Import the random library
from random import randint

# Generate a random integer between 1 and 10 inclusive
num = randint(1, 10)

# Prompt and get input from the user
guess = int(input("Enter your guess between 1 and 10: "))

# Is the guess correct?
if guess == num:
    # The guess is correct
    print("You got it!")
else:
    # The guess is incorrect
    print(f"Sorry, the number is {num}")

# This last statement is always executed,
# after the if statement are complete
print("Done")
```

Example program run:

```
Enter your guess between 1 and 10: 5
Sorry, the number is 3
Done
```

String Comparison

Using comparison operators works for strings as well.

```
truck = 'Dodge'
if truck == 'dodge':
   print('You have a Dodge truck.')
else:
   print('You don't have a Dodge truck.')
```

```
You don't have Dodge truck.
```

Python, like many programming languages, is case sensitive. Dodge is not the same as dodge. Use the `.lower()` operator to make sure you are comparing strings properly.

```
truck = 'Dodge'
if truck.lower() == 'dodge':
   print('You have a Dodge truck.')
else:
   print('You don't have a Dodge truck.')
```

```
You have a Dodge truck.
```

if else

A second form of the **if** statement is alternative execution, in which there are two possibilities. The condition determines which one gets executed.

The **if** statement shown earlier tells the program to test for a condition. If the condition is True, do something special and then continue business as usual. If the condition is False, don't do anything special -- just continue business as usual.

Sometimes our decision processes are more complicated than that. Sometimes we need the program to test for a condition and if the condition is True, the program should do something special. However, if the condition is False, the program should do something different that is also special. After one or the other of the special things is done, the program should continue with business as usual. That is the purpose of the if...else statement.

Pseudo-code for if else Statement

```
if expression is True:
   Execute statements at same indentation level
else:
   Execute different statements at same indentation level
Execute next statement
```

As before, the expression must be one that will evaluate to either True or False.

1. If the expression evaluates to True, the indented statement(s) that follow the expression will be executed in sequence and the indented statement(s) that follow else: will be skipped.

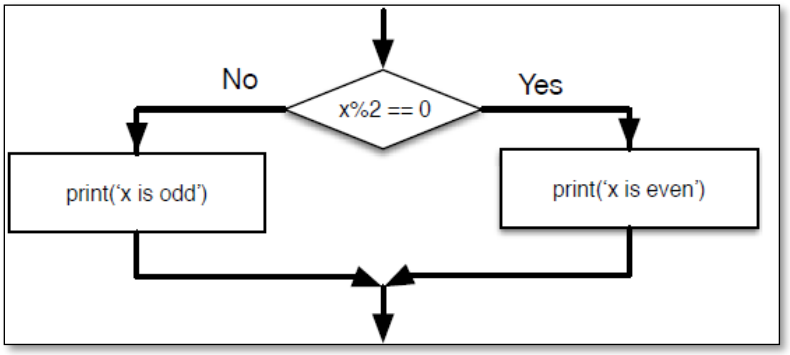
2. If the expression evaluates to False, the indented statement(s) that follow the expression will be skipped and the indented statements that follow else: will be executed.

3. After that, the next statement following the if statement will be executed

The syntax looks like this:

```
if x % 2 == 0:
   print('x is even')
else:
   print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called branches because they are branches in the flow of execution.

Here is an example of a loop combined with if else and the random module.

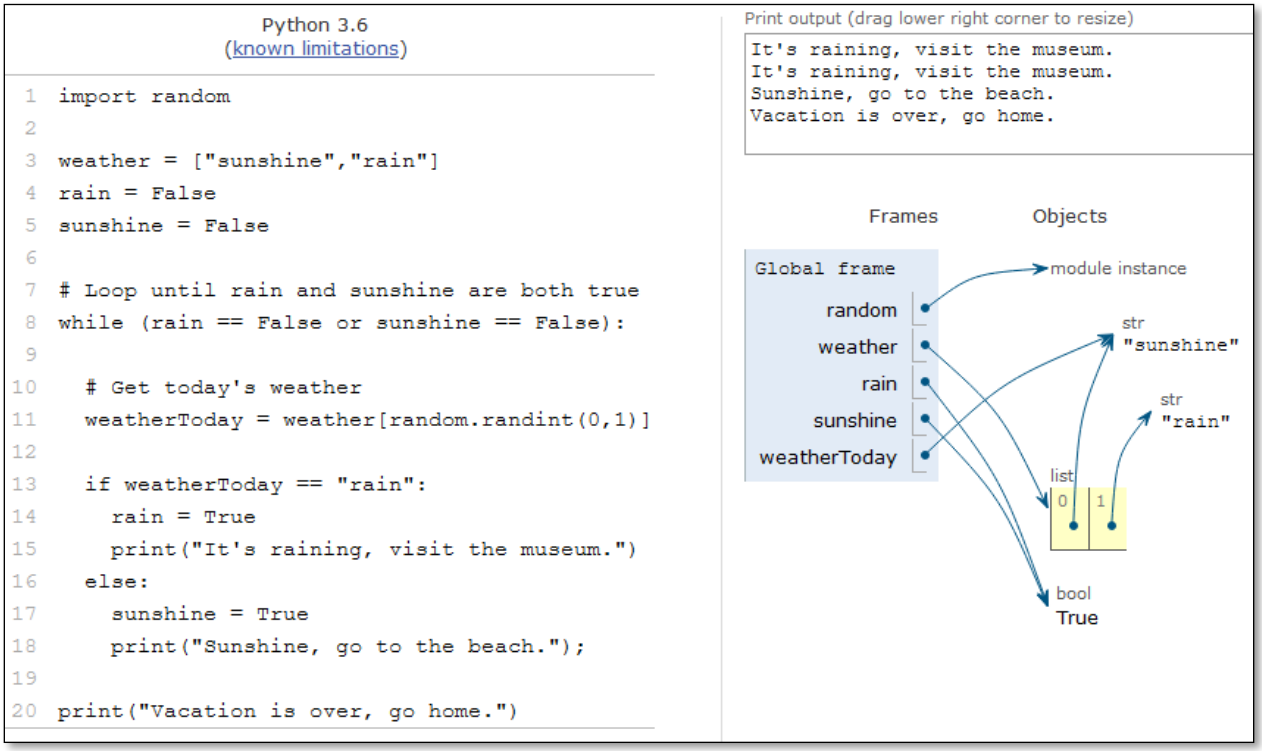
```
import random
# List of weather conditions
weather = ["sunshine", "rain"]
rain = False
sunshine = False

# Loop until rain and sunshine are both true
while (rain == False or sunshine == False):

  # Get today's weather
  weather_today = weather[random.randint(0,1)]

  if weather_today == "rain":
    rain = True
    print("It's raining, visit the museum.")
  else:
    sunshine = True
    print("Sunshine, go to the beach.");

print("Vacation is over, go home.")
```



The code is simulating a short vacation. You want to visit the museum at least once and go to the beach at least once before you go home.

1. The code begins by importing the random module as described earlier. Then it creates and populates three working variables that will be used later.
2. A while loop is used is used to assure that you visit both the museum and the beach at least once before going home. Note the use of the equality operator (==) and the logical (or) operator in the conditional clause of the while loop.
3. The first statement inside the while loop uses the random number generator to get the weather for the day from the list named weather. The result will be either "sunshine" or "rain".
4. Following that, an if...else statement is used to decide whether to visit the museum or to go to the beach based on the weather. Note the use of the equality operator (==) in the conditional clause of the if statement. Also note that the corresponding working variable (rain or sunshine) is set to True to confirm the visit for the benefit of the conditional clause of the while loop.
5. When the museum and the beach have each been visited at least once, the while loop will terminate causing the "go home" print statement to be executed.

Because of the use of a random number generator to control the weather, this program will produce a different output each time you run it.

if elif else (Nested if Statements)

Here are the three keywords that can be used with nested **if** statements:

- if
- else
- elif

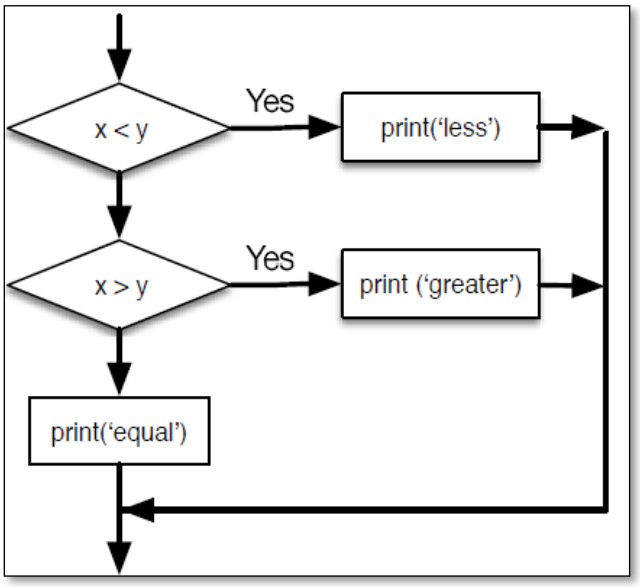
You already know about **if** and **else**. The keyword **elif** is short for *"else if"*. Nested **if** statements can contain zero or more **elif** parts and the **else** part is optional.

The **if...**else construct allows the program to choose between two options. The use of elif makes it possible to write code that can choose among three or more options.

Sometimes there are more than two possibilities, we need more than two branches. One way to express a computation like that is a chained conditional:

```
if x < y:
   print('x is less than y')
elif x > y:
   print('x is greater than y')
else:
   print('x and y are equal')
```

`elif` is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit on the number of `elif` statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.



Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

The program shown below extends the vacation analogy to three options:

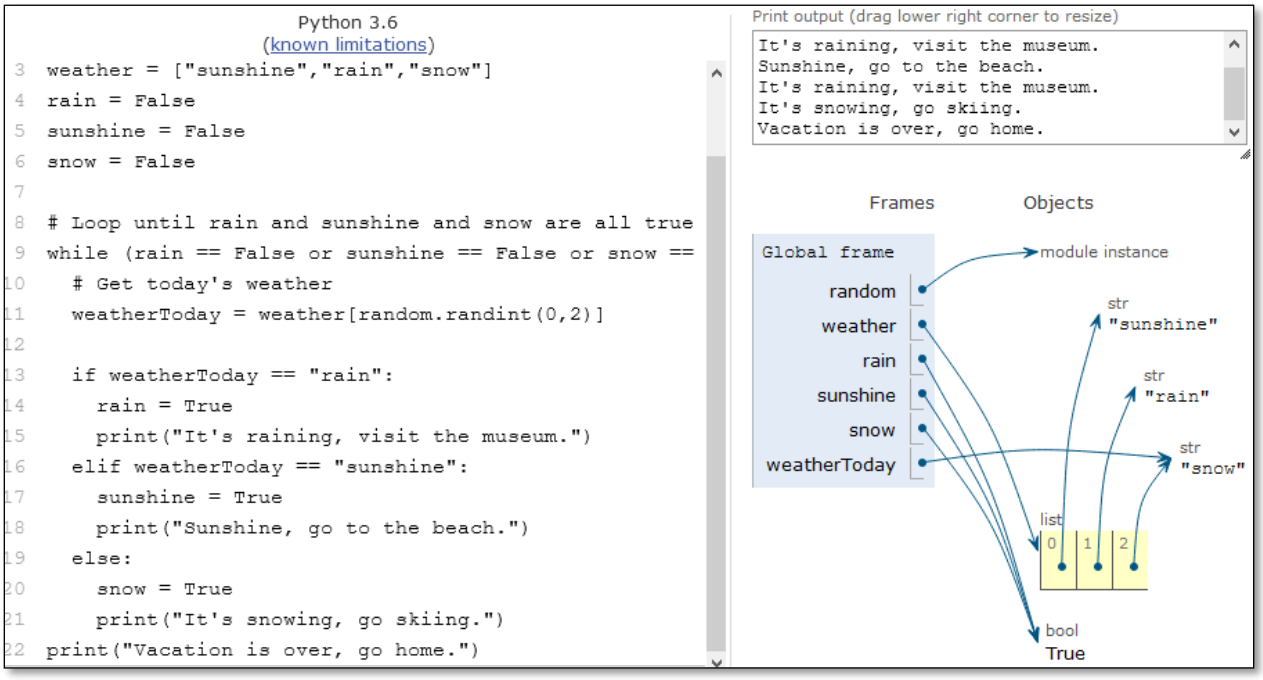
- rain -- visit the museum
- sunshine -- go to the beach
- snow -- go skiing

Once again, you won't go home until you have at least one opportunity to do each of the three activities in the above list.

```
import random
# List of weather conditions
weather = ["sunshine", "rain", "snow"]
rain = False
sunshine = False
snow = False

# Loop until rain and sunshine and snow are all true
while (rain == False or sunshine == False or snow == False):
  # Get today's weather
  weather_today = weather[random.randint(0,2)]

  if weather_today == "rain":
    rain = True
    print("It's raining, visit the museum.")
  elif weather_today == "sunshine":
    sunshine = True
    print("Sunshine, go to the beach.")
  else:
    snow = True
    print("It's snowing, go skiing.")
print("Vacation is over, go home.")
```



Tutorial 3.2 – Grades

This is a simple use of an **if** statement is to assign letter grades. Suppose that scores 90 and above are A's, scores in the 80s are B's, 70s are C's, 60s are D's, and anything below 60 is an F. Here is one way to do this:

```
# Less effecient decision structure

grade = int(input("Enter your score: "))

if grade >= 90:
    print("A")
elif grade >= 80 and grade < 90:
    print("B")
elif grade >= 70 and grade < 80:
    print("C")
elif grade >= 60 and grade < 70:
    print("D")
else:
    print("F")
```

The code above is straightforward, and it works. A more elegant and simpler way to do it is shown below.

Create and save the following program as **grades.py**.

```
"""
    Name: grades.py
    Author:
    Created:
    Purpose: Determine a letter grade from a score
"""

# Get score from the user
score = int(input("Enter your score: "))

# Determine what the grade is and display it
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
elif score >= 60:
    print("D")
else:
    print("F")
```

Sample run:

```
Enter your score: 87
B
```

With the separate **if** statements, each condition is checked regardless of whether it really needs to be. That is, if the score is a 95, the first program will print an A but then continue on and check to see if the score is a B, C, etc., which is a bit of a waste.

Using **elif**, as soon as we find where the score matches, we stop checking conditions and skip all the way to the end of the whole block of statements.

```
elif grade >= 80 and grade < 90:
    print("B")
```

When using **elif**, the second part of the first programs if statement condition, **grade < 90**, becomes unnecessary because the corresponding **elif** does not have to worry about a score of 90 or above, as such a score would have already been caught by the first if statement.

Tutorial 3.3 – Magic 8-Ball

Time for a game break. The Magic 8-Ball knows all and sees all. Find all the answers to life, the universe, and everything.

This game only uses 8 answers. Here are all the standard Magic 8-Ball answers if you decide to expand this tutorial.

It is certain.	As I see it, yes.	Reply hazy, try again.	Don't count on it
It is decidedly so.	Most likely.	Ask again later.	My reply is no.
Without a doubt.	Outlook good.	Better not tell you now.	My sources say no.
Yes – definitely.	Yes.	Cannot predict now.	Outlook not so good.
You may rely on it.	Signs point to yes.	Concentrate and ask again.	Very doubtful.

Create and test a program named **magic_8_ball.py**.

```
"""
    Name: magic_8_ball.py
    Author:
    Created:
    Purpose: Magic 8-Ball game
"""

# Import the random module for random numbers
import random

# Boolean loop variable
answer = True

# A game loop keeps going forever, answer will always be true
while answer:
    question = input("Ask the magic 8 ball a question: (Enter to quit)")

    # Pick a random integer 1-8 inclusive
    answers = random.randint(1, 8)

    # If the user presses the Enter key, exit
    if question == "":
        break
    elif answers == 1:
        print("It is certain")
    elif answers == 2:
        print("Outlook good")
    elif answers == 3:
        print("You may rely on it")
    elif answers == 4:
        print("Ask again later")
    elif answers == 5:
        print("Concentrate and ask again")
    elif answers == 6:
        print("Reply hazy, try again")
    elif answers == 7:
        print("My reply is no")
    elif answers == 8:
        print("My sources say no")
```

Example run:

```
Ask the magic 8 ball a question: (Enter to quit)What is my fortune
Ask again later
Ask the magic 8 ball a question: (Enter to quit)Will I be wealthy?
Reply hazy, try again
Ask the magic 8 ball a question: (Enter to quit)
>>>
```

Nested Conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
   print("x and y are equal"
else:
   if x < y:
      print("x is less than y")
   else:
      print("x is greater than y")
```

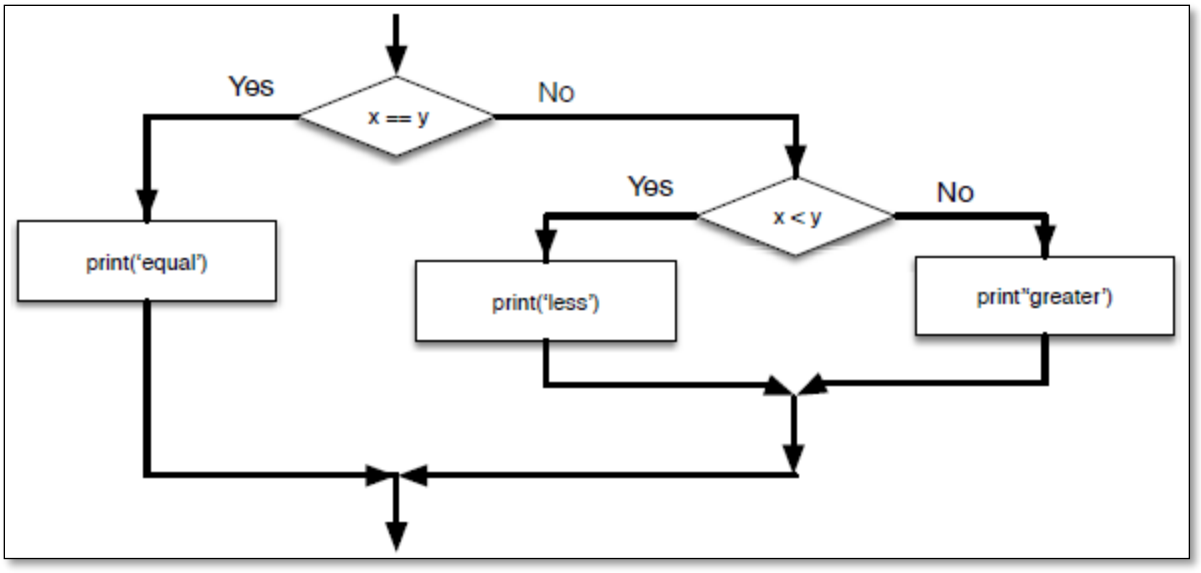
The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
   if x < 10:
      print("x is a positive single-digit number.")
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:




```
if 0 < x and x < 10:
   print("x is a positive single-digit number. ")
```

Catching Exceptions using Try and Except

Earlier we saw a code segment where we used the input and int functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = "What is the air velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What is the air velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think "oops", and move on to our next statement.

If you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Tutorial 3.4 – Fahrenheit to Celsius

Create and test a program named **fahrenheit_to_celsius.py**

Note: You can get the ° degree from any number of websites.

```python
"""
    Name: fahrenheit_to_celsius.py
    Author:
    Created:
    Purpose: Convert Fahrenheit input to Celsius
"""

# Get Fahrenheit temperature from user
fahrenheit = float(input("Enter a Fahrenheit temperature: "))

# Calculate the Celsius equivalent
celsius = ((fahrenheit - 32.0) * 5.0) / 9.0

# Display the Celsius temperature
print(f"{fahrenheit:.2f}° Fahrenheit is equal to {celsius:.2f}° Celcius.")
```

Example program runs:

```
Enter a Fahrenheit temperature: -40
-40.00° Fahrenheit is equal to -40.00° Celcius.
```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
Enter a Fahrenheit temperature: Barney
Traceback (most recent call last):
  File "Z:\_WNCC\Python\Assignments\03 Conditional Execution\Chapter.03 Fahrenhe
it to Celsius.py", line 8, in <module>
    fahrenheit = float(input('Enter a Fahrenheit temperature: '))
ValueError: could not convert string to float: 'Barney'
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except". The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the **try** and **except** feature in Python as an "insurance policy" on a sequence of statements.

Tutorial 3.5 – Fahrenheit to Celsius Try Except

We rewrite our temperature converter as follows:

```
"""
    Name: fahrenheit_to_celsius_try.py
    Author:
    Created:
    Purpose: Use try except to handle improper input
"""

# Try to execute the program
try:
    # Get Fahrenheit temperature from user
    fahrenheit = float(input("Enter a Fahrenheit temperature: "))

    # Calculate the Celsius equivalent
    celsius = ((fahrenheit - 32.0) * 5.0) / 9.0

    # Display the Celsius temperature
    print(f"{fahrenheit:.2f}° Fahrenheit is equal to {celsius:.2f}° Celcius.")

# Catch and handle a program exception
except:
    # Display an error message
    print("Please enter a number next time.")
    # raise is for troubleshooting a program that has an exception
    # raise shows the stack trace as if the try except block didn't exist
    # raise
```

Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

```
Enter a Fahrenheit temperature: Barney
Please enter a number next time.
```

Handling an exception with a try statement is called catching an exception. In this example, the except clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Short-Circuit Evaluation of Logical Expressions

When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called short-circuiting the evaluation.

While this may seem like a fine point, the short-circuit behavior leads to a clever technique called the guardian pattern. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error. But the first and the second examples did not fail because the first part of these expressions x >= 2 evaluated to False so the (x/y) was not ever executed due to the short-circuit rule and there was no error.

We can construct the logical expression to strategically place a guard evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x / y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x / y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In the first logical expression, `x >= 2` is False so the evaluation stops at the and. In the second logical expression, `x >= 2` is True but `y != 0` is False so we never reach (x/y).

In the third logical expression, the `y != 0` is after the (x/y) calculation so the expression fails with an error.

In the second expression, we say that `y != 0` acts as a guard to insure that we only execute (x/y) if y is non-zero.

Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was
- Where it occurred

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible; we are used to ignoring them.

```
File "<stdin>", line 1
y = 6
^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

Glossary

body The sequence of statements within a compound statement.

boolean expression An expression whose value is either True or False.

branch One of the alternative sequences of statements in a conditional statement.

chained conditional A conditional statement with a series of alternative branches.

comparison operator One of the operators that compares its operands: ==, !=,

>, <, >=, and <=.

conditional statement A statement that controls the flow of execution depending on some condition.

condition The Boolean expression in a conditional statement that determines which branch is executed.

compound statement A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

guardian pattern Where we construct a logical expression with additional comparisons to take advantage of the short-circuit behavior.

logical operator One of the operators that combines Boolean expressions: and, or, and not.

nested conditional A conditional statement that appears in one of the branches of another conditional statement.

traceback A list of the functions that are executing, printed when an exception occurs.

short circuit When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.